Bitwise Data Parallelism in Regular Expression Matching

Subtitle Text, if any

Robert D. Cameron Kenneth S. Herdy Dan Lin Meng Lin Ben Hull Thomas S. Shermer Arrvindh Shriraman

Simon Fraser University

{cameron,ksherdy,lindanl,linmengl,bhull,shermer,ashriram}@cs.sfu.ca

Abstract

New parallel algorithms for the classical grep (global regular expression print) problem are introduced together with implementations using commodity SIMD and GPU technologies. Building on the bitwise data parallelism underlying previous work in Unicode transcoding and XML parsing, the new algorithms add the important element of nondeterminism for tackling the full generality of regular expression processing. On widely-deployed commodity hardware using 128-bit SSE2 SIMD technology, our algorithm implementations can substantially outperform traditional grep implementations based on NFAs, DFAs or backtracking. The algorithms are also designed to scale with the availability of additional parallel resources such as the wider SIMD facilities (256-bit) of Intel AVX2 or future 512-bit extensions. Our GPU implementations show further acceleration limited only by data transfer speed.

Categories and Subject Descriptors Theory of computation [*Formal languages and automata theory*]: Regular languages; Computer systems organization [*Parallel architectures*]: Single instruction, multiple data

Keywords regular expression matching, grep, parallel bit stream technology

1. Introduction

The use of regular expressions to search texts for occurrences of string patterns has a long history and remains a pervasive technique throughout computing applications today. The origins of regular expression matching date back to automata theory developed by Kleene in the 1950s [9]. Thompson [22] is credited with the first construction to convert regular expressions to nondeterministic finite automata (NFA). Following Thompson's approach, a regular expression of length m is first converted to an NFA with O(m) nodes. It is then possible to search a text of length n using the NFA in worst case O(mn) time. Often, a more efficient choice is to convert the NFA into a DFA. A DFA has only a single active state at any time in the matching process and hence it is possible to search a text at of length n in worst-case O(n) optimal. However, it is well known that the conversion of an NFA to an equivalent DFA may result in state explosion. That is, the number of resultant DFA states may increase exponentially. In [3] a new approach to text searching was proposed based on bit-parallelism [2]. This technique takes advantage of the intrinsic parallelism of bitwise operations within a computer word. Given a w-bit word, the Shift-Or algorithm [3] algorithm uses the bit-parallel approach to simulate an NFA in O(nm/w) worst-case time.

A disadvantage of the bit-parallel Shift-Or pattern matching approach in comparison to simple string matching algorithms is an inability to skip input characters. For example, the Boyer-Moore family of algorithms [5] skip input characters to achieve sublinear times in the average case. Backward Dawg Matching (BDM) string matching algorithms [7] based on suffix automata are able to skip characters. The Backward Nondeterministic Dawg Matching (BNDM) pattern matching algorithm [25] combines the bit-parallel advantages of Shift-Or and with the character skipping advantages of the BDM algorithm. The nrgrep pattern matching tool is built over the BNDM algorithm, and hence the name nrgrep [13].

a brief review There has been considerable interest in using parallelization techniques to improve the performance of regular expression matching on parallel hardware such as multi-core processors (CPUs), graphics processing units (GPUs), field-programmable gate arrays (FPGAs), and even more exotic architectures such as the Cell Broadband Engine (Cell BE). Scarpazza and Braudaway [21] demonstrated that text processing algorithms that exhibit irregular memory access patterns can be efficiently executed on multicore hardware. In related work, Pasetto et al. presented a flexible tool that performs small-ruleset regular expression matching at a rate of 2.88 Gbps per chip on Intel Xeon E5472 hardware [15]. Naghmouchi et al. demonstrated that the Aho-Corasick (AC) string matching algorithm [1] is well suited for parallel implementation on multi-core CPUs, GPUs and the Cell BE [12, 18]. On each hardware, both thread-level parallelism (additional cores) and data-level parallelism (wide SIMD units) are leveraged for performance. Salapura et. al., advocated the use of vector-style processing for regular expressions in business analytics applications and leveraged the SIMD hardware available on multi-core processors to acheive a speedup of better than 1.8 over a range of data sizes of interest [16]. In [19], Scarpazza and Russell presented a SIMD tokenizer that delivered 1.001.78 Gbps on a single Cell BE chip and extended this approach for emulation on the Intel Larrabee instruction set [17]. On the Cell BE, Scarpazza [20] described a pattern matching implementation that delivered a throughput of 40 Gbps for a small dictionary of approximately 100 patterns, and a throughput of 3.3-3.4 Gbps for a larger dictionary of thousands of patterns. Iorio and van Lunteren [8] presented a string matching implementation for automata that achieves 4 Gbps on the Cell BE. In more recent work, Tumeo et al. [23] presented

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

PPoPP 2014, February 15-19, 2013, Orland, Florida, United States.
Copyright © 2013 ACM 978-1-nnnn-nnnn-n/yy/mm. . . $15.00.
http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

a chunk-based implementation of the AC algorithm for accelerating string matching on GPUs. Lin et al., proposed the Parallel Failureless Aho-Corasick (PFAC) algorithm to accelerate pattern matching on GPU hardware and achieved 143 Gbps throughput, 14.74 times faster than the AC algorithm performed on a four core multi-core processor using OpenMP [10].

Whereas the existing approaches to parallelization have been based on adapting traditional sequential algorithms to emergent parallel architectures, we introduce both a new algorithmic approach and its implementation on SIMD and GPU architectures. This approach relies on a bitwise data parallel view of text streams as well as a surprising use of addition to match runs of characters in a single step. The closest previous work is that underlying bit-parallel XML parsing using 128-bit SSE2 SIMD technology together with a parallel scanning primitive also based on addition [6]. However, in contrast to the deterministic, longest-match scanning associated with the ScanThru primitive of that work, we introduce here a new primitive MatchStar that can be used in full generality for nondeterministic regular expression matching. We also introduce a long-stream addition technique involving a further application of MatchStar that enables us to scale the technique to n-bit addition in $\lceil \log_{64} n \rceil$ steps. We ultimately apply this technique, for example, to perform synchronized 4096-bit addition on GPU warps of 64 threads.

There is also a strong keyword match between the bit-parallel data streams used in our approach and the bit-parallelism used for NFA state transitions in the classical algorithms of Wu and Manber [24], Baez-Yates and Gonnet [2] and Navarro and Raffinot [14]. However those algorithms use bit-parallelism in a fundamentally different way: representing all possible current NFA states as a bit vector and performing parallel transitions to a new set of states using table lookups and bitwise logic. Whereas our approach can match multiple characters per step, bit-parallel NFA algorithms proceed through the input one byte at a time. Nevertheless, the agrep [24] and nrgrep [13] programs implemented using these techniques remain among the strongest competitors in regular expression matching performance, so we include them in our comparative evaluation.

The remainder of this paper is organized as follows. Section 2 briefly describes regular expression notation and the grep problem. Section 3 presents our basic algorithm and MatchStar using a model of arbitrary-length bit-parallel data streams. Section 4 discusses the block-by-block implementation of our techniques including the long stream addition techniques for 256-bit addition with AVX2 and 4096-bit additions with GPGPU SIMT. Section 5 Section 6 Section 7 Section 8 Section 10 concludes the paper with a discussion of areas for future work.

2. Regular Expression Notation and Grep

We follow common Posix notation for regular expressions. A regular expression specifies a set of strings through a pattern notation. Individual characters normally stand for themselves, unless they are one of the special characters `*+?[{\(|^$.` that serve as metacharacters of the notation system. Thus the regular expression `cat` is a pattern for the set consisting of the single 3-character string "`cat`". The special characters must be escaped with a backslash to prevent interpretation as metacharacter, thus `\$` represents the dollar-sign and `\\\\` represent the string consisting of two backslash characters. Character class bracket expressions are pattern elements that allow any character in a given class to be used in a particular context. For example, `[@#%]` is a regular expression that stands for any of the three given symbols. Contiguous ranges of characters may be specified using hyphens; for example `[0-9]` for digits and `[A-Za-z0-9_]` for any alphanumeric character or underscore. If the caret character immediately follows the opening bracket, the class is negated, thus `[^0-9]` stands for any character except a digit. The period metacharacter `.` stands for the class of all characters.

Consecutive pattern elements stand for strings formed by concatenation, thus `[cd][ao][tg]` stands for the set of 8 three-letter strings "`cat`" through "`dog`". The alternation operator `|` allows a pattern to be defined to have to alternative forms, thus `cat|dog` matches either "`cat`" or "`dog`". Concatenation takes precedence over alternation, but parenthesis may be used to change this, thus `(ab|cd)[0-9]` stands for any digit following one of the two prefixes "`ab`" or "`cd`".

Repetition operators may be appended to a pattern to specify a variable number of occurrences of that pattern. The Kleene `*` specifies zero-or-more occurrences matching the previous pattern, while `+` specifies one-or more occurrences. Thus `[a-z][a-z]*` and `[a-z]+` both specify the same set: strings of at least one lower-case letter. The postfix operator `?` specifies an optional component, i.e., zero-or-one occurrence of strings matching the element. Specific bounds may be given within braces: `(ab){3}` specifies the string "`ababab`", `[0-9A-Fa-f]{2,4}` specifies strings of two, three or four hexadecimal digits, and `[A-Z]{4,}` specifies strings of at least 4 consecutive capital letters.

The grep program searches a file for lines containing matches to a regular expression using any of the above notations. In addition, the pattern elements `^` and `$` may be used to match respectively the beginning or the end of a line. In line-based tools such as grep, `.` matches any character except newlines; matches cannot extend over lines. Normally, grep prints all matching lines to its output. However, grep programs typically allow a command line flag such as `-c` to specify that only a count of matching lines be produced; we use this option in our experimental evaluation to focus our comparisons on the performance of the underlying matching algorithms.

3. Matching with Bit-Parallel Data Streams

Whereas the traditional approaches to regular expression matching using NFAs, DFAs or backtracking all rely on a byte-at-a-time processing model, the approach we introduce in this paper is based on quite a different concept: a data-parallel approach to simultaneous processing of data stream elements. Indeed, our most abstract model is that of unbounded data parallelism: processing all elements of the input data stream simultaneously. In essence, we view data streams as (very large) integers. The fundamental operations we apply are based on bitwise logic and long-stream addition.

Depending on the available parallel processing resources, an actual implementation may divide an input stream into blocks and process the blocks sequentially. Within each block all elements of the input stream are processed together, relying the availability of bitwise logic and addition scaled to the block size. On commodity Intel and AMD processors with 128-bit SIMD capabilities (SSE2), we typically process input streams 128 bytes at a time. In this case, we rely on the Parabix tool chain [11] to handle the details of compilation to block-by-block processing. As we show later, however, we have also adapted Parabix technology to processing blocks of 4K bytes at time in our GPGPU implementation, relying on the application of our long-stream addition technique to perform 4096-bit additions using 64 threads working in lock-step SIMT fashion each on 64-bit processors.

A key concept in this streaming approach is the derivation of bit streams that are parallel to the input data stream, i.e., in one-to-one correspondence with the data element positions of the input streams. Typically, the input stream is a byte stream comprising the 8-bit character code units of a particular encoding such as extended ASCII, ISO-8859-1 or UTF-8. However, the method may also easily be used with wider code units such as the 16-bit code

units of UTF-16. In the case of a byte stream, the first step is to transpose the byte stream into eight parallel bit streams, such that bit stream i comprises the ith bit of each byte. These streams form a set of basis bit streams from which many other parallel bit streams can be calculated, such as character class bit streams such that each bit j of the stream specifies whether character j of the input stream is in the class or not. Figure **??** shows an example of an input byte stream in ASCII, the eight basis bit streams of the transposed representation, and several character class bit streams that may be computed from the basis bit streams using bitwise logic. Transposition and character class construction are straightforward using the Parabix tool chain [11].

Marker Streams. Now consider how bit-parallel data streams can be used in regular expression matching. Consider the problem of searching the input stream of Figure **??** to finding occurrence of strings matching the regular expression `a[0-9]*z`. The matching process involves the concept of *marker streams*, that is streams that mark the positions of current matches during the overall process. In this case there are three marker streams computed during the match process, namely, M_1 representing match positions after an initial `a` character has been found, M_2 representing positions reachable from positions marked by M_1 by further matching zero or more digits (`[0-9]*`) and finally M_3 the stream marking positions after a final `z` has been found. Without describing the details of how these streams are computed for the time being, Figure **??** shows what each of these streams should be for our example matching problem. Note our convention that a marker stream contains a 1 bit at the next character position to be matched, that is, immediately past the last position that was matched.

MatchStar. MatchStar takes a marker bitstream and a character class bitstream as input. It returns all positions that can be reached by advancing the marker bitstream zero or more times through the character class bitstream.

Figure **??** illustrates the MatchStar method. The second and third rows are the input bitstreams: the initial marker position bitstream and the character class bitstream derived from the source data.

In the first operation (T_0), marker positions that cannot be advanced are temporarily removed from consideration by masking off marker positions that aren't character class positions using bitwise logic. Next, the temporary marker bitstream is added to the character class bitstream. T_1 has 1s in three types of positions. There will be a 1 immediately following a block of character class positions that spanned one or more marker positions, at any character class positions that weren't affected by the addition (and are not part of the desired output), and at any marker position that wasn't the first in its block of character class positions. Any character class positions that have a 0 in T_1 were affected by the addition and are part of the desired output. These positions are obtained and the undesired 1 bits are removed by XORing with the character class stream. T_2 is now only missing marker positions that were removed in the first step as well as marker positions that were 1s in T_1. The output marker stream is obtained by ORing T_2 with the initial marker stream.

In general, given a marker stream M and a character class stream C, the operation of MatchStar is defined by the following equation.

$$\text{MatchStar}(M, C) = (((M \wedge C) + C) \oplus C) | M$$

Given a set of initial marker positions, the result stream marks all possible positions that can be reached by 0 or more occurrences of characters in class C from each position in M.

4. Block-at-a-Time Processing

The unbounded stream model of the previous section must of course be translated an implementation that proceeds block-at-a-time for realistic application. In this, we primarily rely on the Pablo compiler of the Parabix toolchain [11]. Given input statements expressed as arbitrary-length bitstream equations, Pablo produces block-at-a-time C++ code that initializes and maintains all the necessary carry bits for each of the additions and shifts involved in the bitstream calculations.

In the present work, our principal contribution to the block-at-a-time model is the technique of long-stream addition described below. Otherwise, we were able to use Pablo directly in compiling our SSE2 and AVX2 implementations. Our GPU implementation required some scripting to modify the output of the Pablo compiler for our purpose.

Long-Stream Addition. The maximum word size for addition on commodity processors is typically 64 bits. In order to implement long-stream addition for block sizes of 256 or larger, a method for propagating carries through the individual stages of 64-bit addition is required. However, the normal technique of sequential addition using add-with-carry instructions, for example, is far from ideal.

We have developed a technique using SIMD or SIMT methods for constant-time long-stream addition up to 4096 bits. We assume the availability of the following SIMD/SIMT operations operating on vectors of f 64-bit fields.

- `simd<64>::add(X, Y)`: vertical SIMD addition of corresponding 64-bit fields in two vectors to produce a result vector of f 64-bit fields.
- `simd<64>::eq(X, -1)` : comparison of the 64-bit fields of `x` each with the constant value -1 (all bits 1), producing an f-bit mask value,
- `hsimd<64>::mask(X)` : gathering the high bit of each 64-bit field into a single compressed f-bit mask value, and
- normal bitwise logic operations on f-bit masks.
- `simd<64>::spread(x)` : distributing the bits of an f bit mask, one bit each to the f 64-bit fields of a vector, and

Given these operations, our method for long stream addition of two $f \times 64$ bit values `X` and `Y` is the following.

1. Form the vector of 64-bit sums of `x` and `y`.

   $$\mathtt{R} = \mathtt{simd<64>::add(X,\ Y)}$$

2. Extract the f-bit masks of `X`, `Y` and `R`.

   $$\mathtt{x} = \mathtt{hsimd<64>::mask(X)}$$

   $$\mathtt{y} = \mathtt{hsimd<64>::mask(Y)}$$

   $$\mathtt{r} = \mathtt{hsimd<64>::mask(R)}$$

3. Compute an f-bit mask of carries generated for each of the 64-bit additions of `X` and `Y`.

   $$\mathtt{c} = (\mathtt{x} \wedge \mathtt{y}) \vee ((\mathtt{x} \vee \mathtt{y}) \wedge \neg \mathtt{r})$$

4. Compute an f-bit mask of all fields of `R` that will overflow with an incoming carry bit. This is the *bubble mask*.

   $$\mathtt{b} = \mathtt{simd<64>::eq(R,\ -1)}$$

5. Determine an f-bit mask identifying the fields of `R` that need to be incremented to produce the final sum. Here we find a new application of MatchStar!

   $$\mathtt{i} = \mathtt{MatchStar(c*2,\ b)}$$

source data	`--142578---125-231-----127--5394---94761205-`
M_0	`.......1......1..1..1...1.............1..1..`
$D =$`[0-9]`	`..111111...111.111.....111..1111...11111111.`
$T_0 = M_0 \wedge D$	`.......1.........1......1.............1..1..`
$T_1 = T_0 + D$	`.1.........1111.......1..1..1111..1...1...1.`
$T_2 = T_1 \oplus D$	`.1111111......1111....111.........1111.111..`
$M_1 = T_2 \mid M_0$	`.1111111......1111..1.111.........11111111..`

Figure 1. Match Star

X	19	31	BA	4C	3D	45	21	F1
Y	22	12	45	B3	E2	16	17	36
R	3B	43	FF	FF	1F	5B	38	27
x	0	0	1	0	0	0	0	1
y	0	0	0	1	1	0	0	0
r	0	0	1	1	0	0	0	0
c	0	0	0	0	1	0	0	1
c*2	0	0	0	1	0	0	1	0
b	0	0	1	1	0	0	0	0
i	0	1	1	1	0	0	1	0
Z	3B	44	0	0	1F	5B	39	27

Figure 2. Long Stream Addition

This is the key step. The mask `c` of outgoing carries must be shifted one position (`c*2`) so that each outgoing carry bit becomes associated with the next digit. At the incoming position, the carry will increment the 64-bit digit. However, if this digit is all ones (as signalled by the corresponding bit of bubble mask `b`, then the addition will generate another carry. In fact, if there is a sequence of digits that are all ones, then the carry must bubble through each of them. This is just MatchStar!

6. Compute the final result `Z`.

$$\mathtt{Z} = \mathtt{simd<64>::add(R,\ simd<64>::spread(i))}$$

Figure 2 illustrates the process. In the figure, we illustrate the process with 8-bit fields rather than 64-bit fields and show all field values in hexadecimal notation. Note that two of the individual 8-bit additions produce carries, while two others produce `FF` values that generate bubble bits. The net result is that four of the original 8-bit sums must be incremented to produce the long stream result.

A slight extension to the process produces a long-stream full adder that can be used in chained addition. In this case, the adder must take an additional carry-in bit `p` and produce a carry-out bit `q`. This may be accomplished by incorporating `p` in calculating the increment mask in the low bit position, and then extracting the carry-out `q` from the high bit position.

$$\mathtt{i} = \mathtt{MatchStar(c*2+p,\ b)}$$

$$\mathtt{q} = \mathtt{i\ >>\ f}$$

As described subsequently, we use a two-level long-stream addition technique in both our AVX2 and GPU implementations. In principle, one can extend the technique to additional levels. Using 64-bit adders throughout, $\lceil \log_{64} n \rceil$ steps are needed for n-bit addition. A three-level scheme could coordinate 64 groups each performing 4096-bit long additions in a two-level structure. However, whether there are reasonable architectures that can support fine-grained SIMT style coordinate at this level is an open question.

Using the methods outlined, it is quite conceivable that instruction set extensions to support long-stream addition could be added for future SIMD and GPU processors. Given the fundamental nature of addition as a primitive and its novel application to regular expression matching as shown herein, it seems reasonable to expect such instructions to become available.

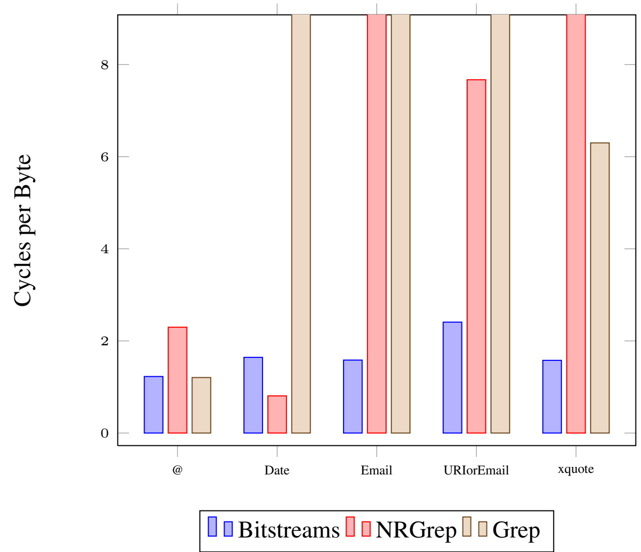

Figure 3. Cycles per Byte

5. Analytical Comparison with DFA and NFA Implementations

1. Operations
2. Memory behaviour per input byte: note tables of DFA/NFA.

   Bille and Throup *Faster regular expression matching[4]*

```
void add_ci_co(bitblock256_t x, bitblock256_t y, carry_t carry_in, carry_t & carry_out, bitblock256_t & sum) {
  bitblock256_t all_ones = simd256<1>::constant<1>();
  bitblock256_t gen = simd_and(x, y);
  bitblock256_t prop = simd_xor(x, y);
  bitblock256_t partial_sum = simd256<64>::add(x, y);
  bitblock256_t carry = simd_or(gen, simd_andc(prop, partial_sum));
  bitblock256_t bubble = simd256<64>::eq(partial_sum, all_ones);
  uint64_t carry_mask = hsimd256<64>::signmask(carry) * 2 + convert(carry_in);
  uint64_t bubble_mask = hsimd256<64>::signmask(bubble);
  uint64_t carry_scan_thru_bubbles = (carry_mask + bubble_mask) &~ bubble_mask;
  uint64_t increments = carry_scan_thru_bubbles | (carry_scan_thru_bubbles - carry_mask);
  carry_out = convert(increments >> 4);
  uint64_t spread = 0x0000200040008001 * increments & 0x0001000100010001;
  sum = simd256<64>::add(partial_sum, _mm256_cvtepu16_epi64(avx_select_lo128(convert(spread))));
}
```

Figure 11. AVX2 256-bit Addition

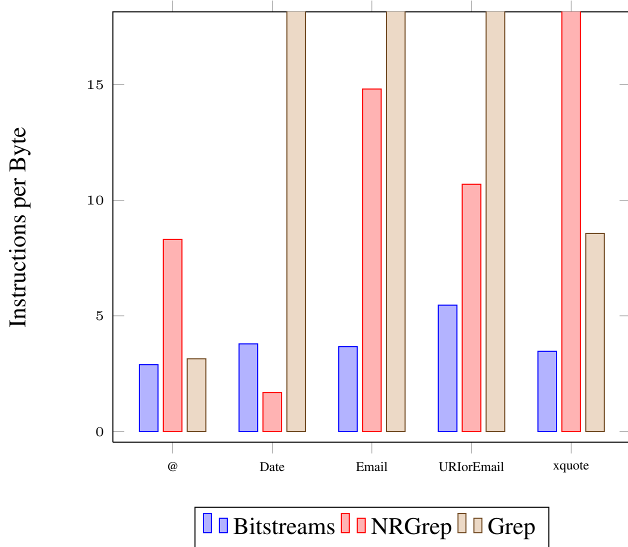

Figure 4. Instructions per Byte

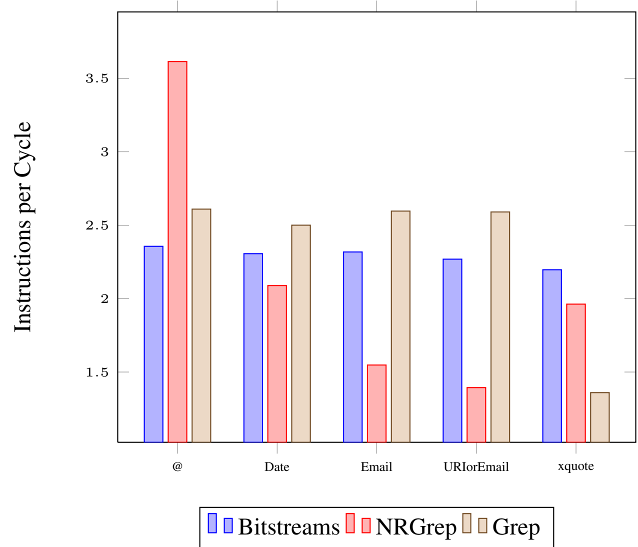

Figure 5. Instructions per Cycle

6. Commodity SIMD Implementation and Experimental Evaluation

6.1 Implementation Notes

6.2 Evaluation Methodology

6.3 Comparison

7. SIMD Scalability

7.1 AVX Stream Addition

8. GPU Implementation

9. Miscellaneous

9.1 Skipping

9.2 Unicode

The introduction of Unicode as a common encoding system including the characters of all the world's written languages and notation systems has introduced some complexity for regular expression matching engines. Nevertheless, most modern tools and libraries do include some form of Unicode support, with varying degrees of performance loss.

UTF-8, UTF-16 and UTF-32 are the three common transformation formats of Unicode depending on whether character code points are encoded using 8-bit, 16-bit or 32-bit code units. In the case of 32-bit code units of UTF-32, each Unicode character is encoded as a single 32-bit unit, with the high 11 bit positions all zero. A stream of UTF-32 encoded text can then be processed using a straightforward application of the techniques described above by first transforming to a set of 21 parallel bit streams for each of the significant bit positions within UTF-32.

For most practical purposes, UTF-16 can also be processed similarly, considering that each 16-bit code unit represents a single character. For the rarely used characters of the Unicode supplementary plane, two 16-bit code units are required. Such a two code unit sequences is known as a surrogate pair. Following the common practice of treating each member of a surrogate pair as pseudo-character, UTF-16 can also be processed by the straightforward transposition to 16 parallel bit streams and application of the techniques above.

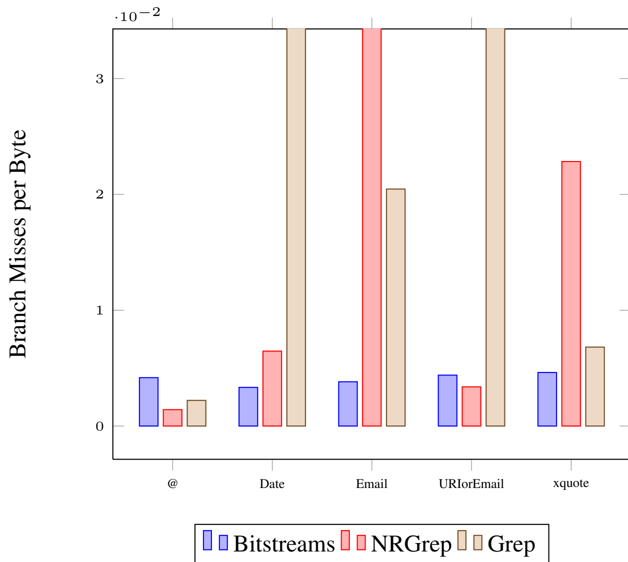

Figure 6. Branch Misses per Byte

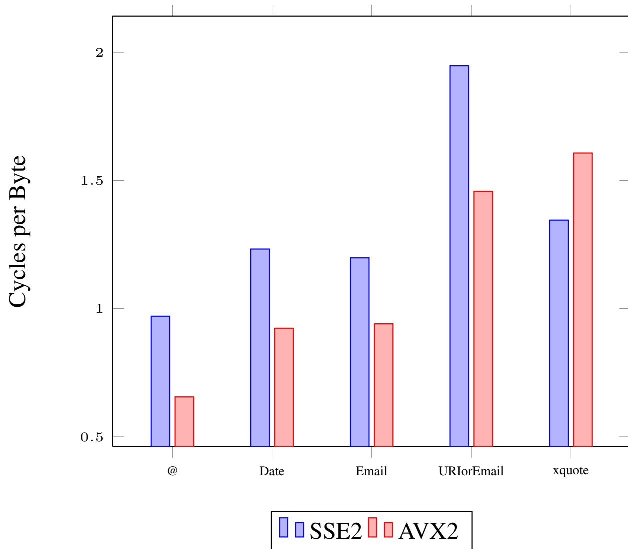

Figure 7. Cycles per Byte

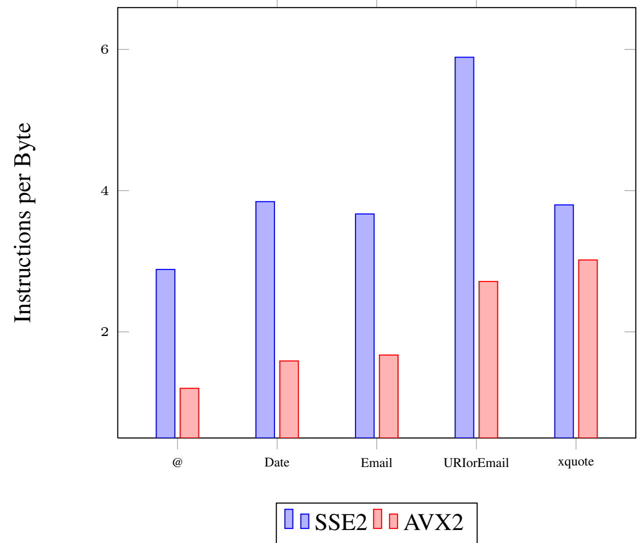

Figure 8. Instructions per Byte

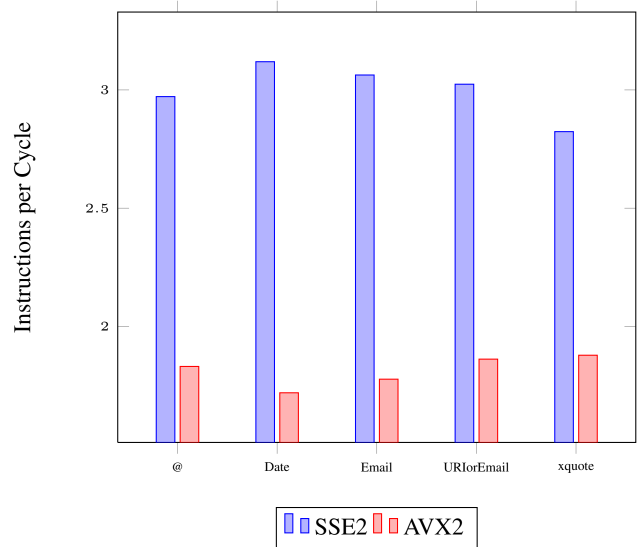

Figure 9. Instructions per Cycle

UTF-8 is probably the most widely used of the Unicode formats, primarily because of its compatibility with widely deployed networking software based on 8-bit extended-ASCII character representations. UTF-8 is a variable length coding system in which each Unicode character is represented using one to four 8-bit code units.

In order to safely process UTF-8, it is necessary to validate that it is well-formed. Each byte is classified as either an ordinary ASCII byte (high bit clear: range 0x00-0x7F), a UTF-8 prefix byte of a 2-, 3- or 4- byte sequence (in ranges 0xC2-0xDF, 0xE0-0xEF, and 0xF0-F4, respectively) or as a UTF-8 suffix byte in the range 0x80-0xBF. Parallel bit stream technology achieves this validation easily, as documented previously for UTF-8 to UTF-16 transcoding.

The UTF-8 byte classification streams produced as a byproduct of validation then enable regular expression on the UTF-8 input streams as follows. For each multibyte character used in the pattern, a character classification bitstream may be formed to identify the occurrence of such characters at each position in the source stream at which the final byte of the character is found. Figure **??** illustrates. (Describe).

Using this approach to multibyte classification, matching of a single k-byte character is straightforward. All current match positions are shifted forward $k - 1$ positions and then combined with the computed character class using bitwise-and. The result is then shifted forward one position to produce the result of the multibyte character match. This method easily extends to character classes comprising multibyte characters all of the same length.

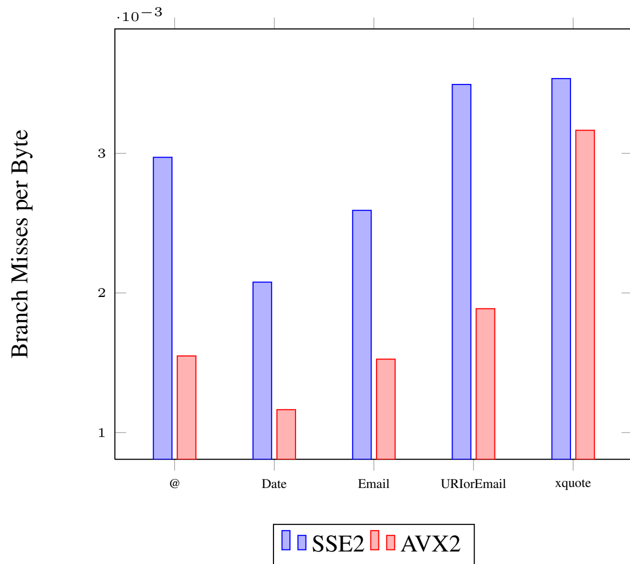

Figure 10. Branch Misses per Byte

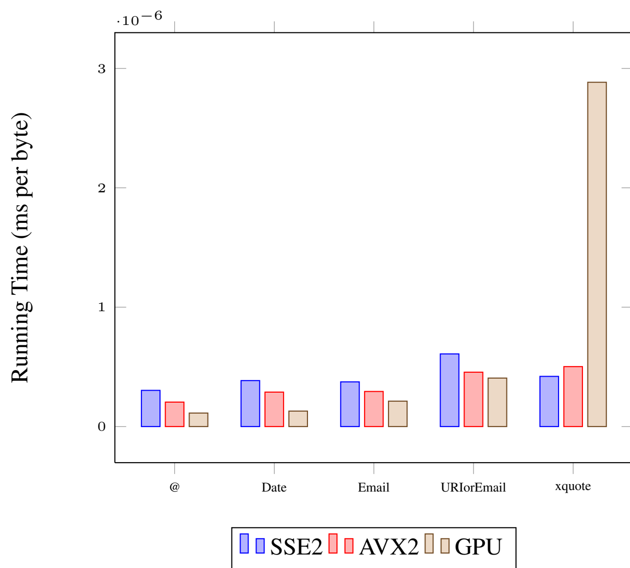

Figure 12. Running Time

Matching a character class comprising characters of different lengths requires a slightly different strategy. In this case, we take advantage of a bitstream `nonfinal` formed from the UTF-8 byte classification streams to consist of all those positions at which a UTF-8 prefix byte is found, as well as those positions at which the second byte of a 3-byte sequence or the second or third byte of a 4-byte sequence are found. We then apply `ScanThru(current,nonfinal)` to advance all current matches to the final position of the next character in the input. Bitwise-and combination with the character class bitstream produces the result identifying matches with this class, the result is then shifted forward 1 position.

The MatchStar operation for matching arbitrary sequences of a character class can similarly take advantage of the `nonfinal` stream. For this case, we form the bitwise-or of the `nonfinal` stream with the character class stream prior to applying MatchStar. The result will propagate bits to the first character position of matches and to final positions of nonmatches. We then clear the nonmatches by combining the result stream with the `suffix` byte stream using bitwise-and.

10. Conclusion

Contributions A new class of regular expression matching algorithm has been introduced based on the concept of bit-parallel data streams together with the MatchStar operation. The algorithm is fully general for nondeterministic regular expression matching; however it does not address the nonregular extensions found in Perl-compatible backtracking implementations. Taking advantage of the SIMD features available on commodity processors, its implementation in a grep too offers consistently good performance in contrast to available alternatives. While lacking some special optimizations found in other engines to deal with repeated substrings or to perform skipping actions based on fixed substrings, it nevertheless performs competitively in all cases. The algorithm tends to scale very well with regular expression complexity, often with order-of-magnitude performance advantage over even the best of its competitors.

A parallelized algorithm for long-stream addition has also been introduced in the paper making a key contribution to the scalability of the bit-parallel matching technique overall and that of MatchStar in particular. This algorithm has enabled straightforward extension of the matching algorithm to the implementation using 256-bit AVX2 technology as well as 4096-bit SIMT implementation on an AMD GPU.

Future Work An important area of future work is to develop and assess multicore versions of the algorithm to handle regular expression matching problems involving larger rulesets than are typically encountered in the grep problem. Such implementations could have useful application in tokenization and network intrusion detection for example.

Another area of interest is to extend the capabilities of the underlying method with addition features for substring capture, zero-width assertions and possibly backreference matching. Extending Unicode support beyond basic Unicode character handling to include full Unicode character class support and normalization forms is also worth investigating.

Acknowledgments

This research was supported by grants from the Natural Sciences and Engineering Research Council of Canada and MITACS, Inc.

References

[1] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Commun.ACM*, 18(6):333–340, June 1975. URL `http://doi.acm.org/10.1145/360825.360855`.

[2] R. Baeza-Yates and G. H. Gonnet. A new approach to text searching. *Communications of the ACM*, 35(10):74–82, 1992.

[3] R. A. Baeza-yates, B. Encalada, and G. H. Gonnet. A new approach to text searching.

[4] P. Bille and M. Thorup. Faster regular expression matching. In *Automata, Languages and Programming*, pages 171–182. Springer, 2009.

[5] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

[6] R. D. Cameron, E. Amiri, K. S. Herdy, D. Lin, T. C. Shermer, and F. P. Popowich. Parallel scanning with bitstream addition: An XML case study. In *Euro-Par 2011 Parallel Processing*, pages 2–13. Springer, 2011.

[7] M. Crochemore, W. Rytter, and M. Crochemore. *Text algorithms*, volume 698. World Scientific, 1994.

[8] F. Iorio and J. V. Lunteren. Fast pattern matching on the cell broadband engine. In *2008 Workshop on Cell Systems and Applications (WCSA), affiliated with the*, 2008.

[9] S. C. Kleene. Representation of events in nerve nets and finite automata. 1951.

[10] C. Lin, C. Liu, L. Chien, and S. Chang. Accelerating pattern matching using a novel parallel algorithm on gpus. *IEEE Transactions on Computers*, 62(10), 2013.

[11] D. Lin, N. Medforth, K. S. Herdy, A. Shriraman, and R. Cameron. Parabix: Boosting the efficiency of text processing on commodity processors. In *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, pages 1–12. IEEE, 2012.

[12] J. Naghmouchi, D. P. Scarpazza, and M. Berekovic. Small-ruleset regular expression matching on gpgpus: quantitative performance analysis and optimization. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 337–348, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0018-6. URL `http://doi.acm.org/10.1145/1810085.1810130`.

[13] G. Navarro. Nr-grep: A fast and flexible pattern matching tool. *Software Practice and Experience (SPE*, 31:2001, 2000.

[14] G. Navarro and M. Raffinot. A bit-parallel approach to suffix automata: Fast extended string matching. In *Combinatorial Pattern Matching*, pages 14–33. Springer, 1998.

[15] D. Pasetto, F. Petrini, and V. Agarwal. Tools for very fast regular expression matching. *Computer*, 43(3):50–58, 2010.

[16] V. Salapura, T. Karkhanis, P. Nagpurkar, and J. Moreira. Accelerating business analytics applications. In *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, pages 1–10. IEEE, 2012.

[17] D. P. Scarpazza. Is larrabee for the rest of us? *Dr.Dobbs J*, 2009.

[18] D. P. Scarpazza. Top-performance tokenization and small-ruleset regular expression matching. *International Journal of Parallel Programming*, 39(1):3–32, 2011.

[19] D. P. Scarpazza and G. F. Russell. High-performance regular expression scanning on the cell/be processor. In *Proceedings of the 23rd international conference on Supercomputing*, pages 14–25. ACM, 2009.

[20] D. P. Scarpazza and G. F. Russell. High-performance regular expression scanning on the cell/be processor. In *Proceedings of the 23rd international conference on Supercomputing*, pages 14–25. ACM, 2009.

[21] D. P. Scarpazza, O. Villa, and F. Petrinni. Fast string searches & multicore processors mapping fundamental algorithms on parallel hardware, 2008.

[22] K. Thompson. Programming techniques: Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, 1968.

[23] A. Tumeo, O. Villa, and D. Sciuto. Efficient pattern matching on gpus for intrusion detection systems. In *Proceedings of the 7th ACM international conference on Computing frontiers*, pages 87–88. ACM, 2010.

[24] S. Wu and U. Manber. agrepa fast approximate pattern-matching tool. *Usenix Winter 1992*, pages 153–162, 1992.

[25] S. Wu and U. Manber. Fast text searching: allowing errors. *Communications of the ACM*, 35(10):83–91, 1992.